

# **Detecting Silent Data Corruption for Extreme-Scale Application through Data Mining**

---

**Mathematics and Computer Science Division**



### **About Argonne National Laboratory**

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see [www.anl.gov](http://www.anl.gov).

### **DOCUMENT AVAILABILITY**

**Online Access:** U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via DOE's SciTech Connect (<http://www.osti.gov/scitech/>)

### **Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):**

U.S. Department of Commerce  
National Technical Information  
Service 5301 Shawnee Rd  
Alexandria, VA 22312  
**[www.ntis.gov](http://www.ntis.gov)**  
Phone: (800) 553-NTIS (6847) or (703) 605-6000  
Fax: (703) 605-6900  
Email: **[orders@ntis.gov](mailto:orders@ntis.gov)**

### **Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):**

U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831-0062  
**[www.osti.gov](http://www.osti.gov)**  
Phone: (865) 576-8401  
Fax: (865) 576-5728  
Email: **[reports@osti.gov](mailto:reports@osti.gov)**

### **Disclaimer**

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

## **Detecting Silent Data Corruption for Extreme-Scale Applications through Data Mining**

---

prepared by:  
Leonardo Bautista-Gomez  
Franck Cappello

Argonne National Laboratory

October 31, 2014

# Detecting Silent Data Corruption for Extreme-Scale Applications through Data Mining

Leonardo Bautista-Gomez and Franck Cappello  
Argonne National Laboratory

October 20, 2014

## Abstract

Supercomputers allow scientists to study natural phenomena by means of computer simulations. Next-generation machines are expected to have more components and, at the same time, consume several times less energy per operation. These trends are pushing supercomputer construction to the limits of miniaturization and energy-saving strategies. Consequently, the number of soft errors is expected to increase dramatically in the coming years. While mechanisms are in place to correct or at least detect some soft errors, a significant percentage of those errors pass unnoticed by the hardware. Such silent errors are extremely damaging because they can make applications silently produce wrong results. In this work we propose a technique that leverages certain properties of high-performance computing applications in order to detect silent errors at the application level. Our technique detects corruption solely based on the behavior of the application datasets and is completely application-agnostic. We propose multiple corruption detectors, and we couple them to work together in a fashion transparent to the user. We demonstrate that this strategy can detect the majority of the corruptions, while incurring negligible overhead. We show that with the help of these detectors, applications can have up to 80% of coverage against data corruption.

## 1 Introduction

High-performance computing (HPC) is changing the way scientists make discoveries. Natural phenomena can be modeled as scientific codes and studied by means of computational simulations. Science applications require ever-larger machines to solve larger problems with higher accuracy. While future machines promise to tackle those complex science problems, they are also raising new challenges. Both the transistor size and the energy consumption of future systems must be significantly reduced, steps that might dramatically impact the soft error rate (SER) according to recent studies [9, 3].

Random memory access (RAM) devices have been intensively protected against soft errors through error-correcting codes (ECCs) because they have the

largest share of the susceptible surface on high-end computers. However, not all parts of the system are ECC protected: in particular, logic units and registers inside the processing units are usually not ECC protected because of the space, time and energy cost that ECC requires to work at low level. Historically, the SER of central processing units was minimized through a technique called *radiation hardening*, which consists of increasing the capacitance of circuit nodes in order to increase the critical charge needed to change the logic level. Unfortunately, this technique involves increasing either the size or the energy consumption of the components, which might be prohibitively expensive at extreme scale. Thus, a non-negligible percentage of soft errors could pass undetected by the hardware, corrupting the numerical data of scientific applications. This is called silent data corruption (SDC).

In this work, we state that the datasets produced by HPC applications have characteristics that reflect the properties of the underlining physical phenomena that those applications seek to model. In particular, we propose to leverage the spatial and temporal behavior of HPC datasets in order to predict the *normal* evolution for the datasets. With this approach, SDCs will *push* the corrupted data point out of the expected range of *normal* values, making it an *outlier*.

The contributions of this work can be summarized as follows:

- We study the propagation of corruption on HPC applications, including the transfer to other processes.
- We propose four SDC detectors that detect anomalies based on the spatial and temporal properties of the datasets.
- We propose a fuzzy logic module that integrates the four detectors and provides the level of anomaly suspicion to the user.
- We implement the four detectors and optimize them to minimize the memory footprint of our technique.
- We evaluate the detection capabilities of our combined detectors and show their with multiple HPC applications.
- We demonstrated that with the help of these detectors, applications can have up to 80% of coverage against data corruption.

The rest of this article is organized as follows. Section 2 presents the related work. Section 3 introduces our proposed detectors and the fuzzy logic module. Section 4 shows our evaluation, including the corruption propagation analysis and the detectors performance. Section 5 concludes this work and present some ideas for future work.

## 2 Related Work

The problem of data corruption for extreme-scale computers has been the target of numerous studies. They can be classified in three groups depending on their

level of generality, that is, how easily a technique can be applied to a wide spectrum of HPC applications. They also have different cost in time, space, and energy. An ideal SDC detection technique should be as general as possible, while incurring a minimum cost over the application.

## 2.1 Hardware-Level Detection

The most general method is to try to solve the problem of data corruption at the hardware level. This method is extremely general because applications do not require any adaptation to benefit from such detectors. Considerable literature exists on soft errors rates [11, 12, 3, 4] and detection techniques at the hardware level [10, 6]. Implementing these techniques efficiently is difficult, however, under the strict constraints of extreme-scale computing (e.g., low power consumption). Even if such requirements could be met, it is unclear whether the market will drive the technologies in this direction.

## 2.2 Process Replication

Process replication has been used for many years to guarantee correctness in critical systems and its application to HPC systems has been studied Fiala et al., for example, have proposed using double-redundant computation to detect SDC by comparing the messages transmitted between the replicated processes [7]. The authors also suggest employing triple redundancy to enable data correction using an voting scheme. This approach is general in that applications need little adaptation to benefit from double or triple redundancy. Unfortunately, double- and triple-redundant computation always imposes large overheads, since the number of hardware resources required double or triple. In addition, the cost of energy consumption is heavily increased when using full replication, not only because of the extra computation, but also because of the extra communications.

## 2.3 Algorithm-Based Fault Tolerance

A promising technique against data corruption is algorithm-based fault tolerance (ABFT) [8]. This technique uses extra checksums in linear algebra kernels in order to detect and correct corruptions [8, 5]. However, ABFT has been implemented only on some linear algebra kernels, which is only a subset of the vast spectrum of computational kernels. Moreover, this technique is not general, since each algorithm needs to be adapted for ABFT. Furthermore, even applications that employ only ABFT-protected kernels could fail to detect SDCs if the corrupted data lies *outside* the ABFT-protected regions.

## 2.4 Approximate Computing

Another type of SDC detection is based on the idea of approximate computing, in which a computing kernel is paired with a cheaper and less accurate kernel that will produce *close enough* results that can be compared with those generated

by the main computational kernel [2]. This detection mechanism has shown promising results but is still not general enough, since each application will need to manually be complemented with the required approximate computing kernels. Furthermore, complex applications will need to adapt multiple different kernels to offer good coverage.

### 3 Unsupervised Anomaly Detection

In this work we propose to use *data mining* to detect SDC during runtime in extreme-scale scientific applications. We believe that a strategy based on *data analytics* is an interesting path to explore for several reasons. First, such an approach is completely independent of the underlying algorithm and therefore dramatically more general than algorithm-based techniques. Second, one can develop lightweight data-monitoring techniques that impose a low overhead on the application compared with that from extremely expensive techniques such as double and triple redundancy. Third, data monitoring and outlier detection can be offered by the runtime in a fashion transparent to the user.

Our main idea is to monitor the application datasets during runtime in order to find behavior patterns and therefore raise alerts when some data points go outside the expected range of *coherent* values. Such inconsistent data points are called *outliers*. This *anomaly detection* strategy has been widely used in multiple domains such as medical analysis. In some domains, large banks of data, including normal and abnormal cases, are available to train the anomaly detectors. The use of such *labeled* data is called *supervised anomaly detection*. In the context of large HPC applications, we propose an *unsupervised anomaly detection* given that in most cases users do not have a large bank of normal and corrupted datasets for training our detectors.

#### 3.1 Distribution Anomaly Detector

The first detector we propose is a detector that simply finds the probability density function (PDF) of the target dataset  $\gamma$  (gamma) at time step  $t^k$ . Then, based on the current PDF and the evolution of the PDFs at previous time steps, it gives a prediction of the possible PDF for the next time step using a low-order approximation. We call this detector a  $\gamma$  - *detector*. (We use a low-order approximation in order to minimize the computation work; we plan to work on the accuracy of our predictions later.) We quantify the efficacy of an SDC detector using two measures: its *recall* and its *precision*. The recall is given by Equation 1 and the precision by Equation 2.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (1)$$

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (2)$$

In this work, we show that this first-order approximation is enough to guarantee a large coverage of the most significant corruptions and that it succeeds in keeping the computation overhead negligible. Any observed data point located outside the boundaries of the predicted PDF will be treated as an outlier. For instance, let us imagine a dataset evolving during the execution and monitored by our detector. The detector will analyze the PDF at each time step and predict an expected range of values for the next time step. As shown in Figure 1(a), the detector predicts that normal values should be inside the segment  $[4.000, 4.050]$ . A data point with the value 4.0312495 is inside that range. If that data point was to be corrupted in the one of the 16 most significant bits of the IEEE floating point representation, the value would automatically move outside the expected range of normal values and it will be detected as an outlier. Now, let us imagine that our detector could give a more accurate prediction, giving  $[4.03115; 4.03185]$  as the interval of normal values. In such case, any corruption in the 24 most significant bits will push the point outside the new and narrower interval. As we can see, the accuracy of the prediction has a direct impact on the detection recall of our detectors.

Another point that should be taken into account while measuring the detection capabilities of these detectors, is that not all the bits in the IEEE floating point representation need to be covered. For instance, a corruption in the most significant bits are likely to generate numerical instability, inducing the application to crash. Such soft errors might be silent to the hardware but not to the application. On the other hand, corruption happening in the least significant bits of the mantissa might produce deviations that are lower than the allowed error of the application, hence, they are negligible. Coming back to the example of our second detector, if we neglected the 4 most significant bits (numerical instability) and the 4 least significant bits (negligible error), we could say that the detector has a coverage of 20 bits out of 24 (83% coverage).

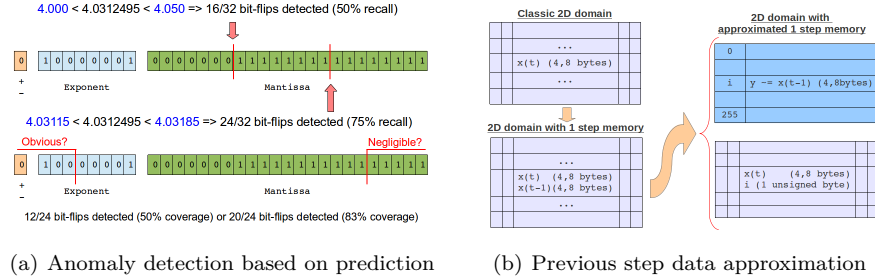


Figure 1: Approximate prediction system of the detectors.

### 3.2 Spatial Anomaly Detector

The second type of detector that we propose is a detector that analyzes the space variations of the dataset. This detector will compute at each time step  $t^k$

a  $\delta$  (delta) field, where  $\delta$  is computed as shown in Equation 3 for a 2D domain.

$$\delta_{(i,j)}^k = \gamma_{(i,j)}^k - \gamma_{(g,h)}^k \quad \text{where } g \in \{i-1; i; i+1\} \wedge h \in \{j-1; j; j+1\} \quad (3)$$

This computation can take into account neighbors in multiple dimensions or in a single dimension, as well as many neighbor points (e.g., 5-point stencil) depending on the preferences of the user. Then, the detector will produce and store the PDF of the  $\delta$  field for the current time step  $t^k$  and predict a PDF for the time step  $t^{k+1}$ . Similarly to the previous case, any point showing a  $\delta_{(i,j)}^{k+1}$  outside the expected range of normal values will be treated as an outlier. We call this detector a  $\delta$  - *detector*.

### 3.3 Temporal Anomaly Detector

The third type of detector that we have developed is based on the temporal evolution of a dataset. More precisely, for each data point we compute the difference  $\epsilon$  (epsilon) as shown in Equation 4.

$$\epsilon_{(i,j)}^k = \gamma_{(i,j)}^k - \gamma_{(i,j)}^{k-1} \quad (4)$$

Then, similarly to the other detectors, we compute the distribution of  $\epsilon$  at time step  $t^k$ , and we compute a low-order approximation of the next PDF of  $\epsilon$ , so that any  $\epsilon_{(i,j)}^{k+1}$  outside the predicted PDF is treated as an outlier. However, computing  $\epsilon^k$  requires saving  $\gamma^{k-1}$  in memory, which involves a large space overhead. To avoid such memory overhead, we sacrifice some accuracy by indexing the PDF (256-bins), so that instead of keeping  $x_{(i,j)}^{k-1}$  for each point of the domain, we keep only a 1-byte index to the nearest value to  $\gamma_{(i,j)}^k - 1$  in the 256-bins PDF, as shown in Figure 1(b). In this way, we reduce the memory footprint of such detector from 4 or 8 bytes (for single or double precision, respectively) to only 1 byte. We call this detector an  $\epsilon$  - *detector*.

### 3.4 Spatiotemporal Anomaly Detector

The fourth detector that we propose in this work is a spatiotemporal detector that computes the time evolution  $\zeta$  (zeta) of the  $\delta$  field computed by  $\delta$  - *detector*, as shown in Equation 5.

$$\zeta_{(i,j)}^k = \delta_{(i,j)}^k - \delta_{(i,j)}^{k-1} \quad (5)$$

Computing the time gradient of the space gradient gives us an idea of when a dataset increases or decreases its level of *turbulence*. Similarly to the temporal anomaly detector, one must keep  $\delta$  values of the previous time step in order to compute the time difference. Thus, we employ the same indexing technique (loosing some accuracy) to reduce the overhead from 4 or 8 bytes to only 1 byte. We call this detector a  $\zeta$  - *detector*.

### 3.5 Fuzzy Logic

We couple the four SDC detectors through a fuzzy logic module. In contrast with a binary detector that would return *True* or *False* to express whether an outlier was detected in the application datasets, fuzzy logic allows us to express how much a data point is an outlier of the dataset. We note that fuzzy logic should not be mistaken with classic probabilities. A probability expresses (in this case) the uncertainty about a data point being an outlier or not, whereas fuzzy logic expresses the degree to which a data point does not belong to a dataset: there is no uncertainty about this degree.

Given that all four detectors are based on a PDF, one can easily quantify how far an outlier is from the prediction and can normalize this deviation for all four detectors. Then, for each data point we can compute an accumulated outlier score including the four detectors. This enables users to take different actions for different levels of anomalies. For instance, a user could choose to ignore detections with a very low degree of deviation and decide to restart from the last checkpoint in the case of an outlier presenting large deviations. In this way, fuzzy logic provides users with more information, thus empowering them to take more accurate actions in the presence of corruption suspicion.

Another important feature that we propose together with this fuzzy detector is the capability to tune the fuzzy logic module, in order to give different weight to the detectors. Although the default method will give same weight to all four detectors, users may want to give more importance to a particular type of detector depending on the application or the input conditions. For instance, when a user increases the time-stepping resolution of a simulation, it is normal to expect lower changes from a time step to another. Therefore it makes sense to give more weight to the  $\epsilon$  - *detector* in such a configuration. The same idea can be applied to the  $\delta$  - *detector* when changing the scale and the space resolution of a simulation. We note that by weighting the different detectors, users can completely remove a detector from the system. For instance, one could remove the  $\zeta$  - *detector* and the  $\epsilon$  - *detector* in order to minimize the memory overhead.

## 4 Evaluation

To evaluate our proposed detectors, we studied one of the most challenging applications in this context: a turbulent flow in a 3D duct modeled as a large eddy simulation using a two-stage time-differencing scheme based on higher accuracy for compressible gas using Navier-Stokes equations. This model of turbulence is well known in the scientific community and is widely used in computational fluid dynamics (CFD). It represents a large set of HPC applications, ranging from weather prediction to aerospace engineering. Turbulence simulations are well known to have a chaotic and hard-to-predict behavior. For this simulation, the 3D duct is divided in  $N$  sections along the length (x axis) of the duct, where  $N$  corresponds to the number of MPI ranks in the simulation.

We implement all the proposed detectors inside the FTI library [1], so that

applications need only to define the datasets to protect; all the detection work will be done automatically in a transparent fashion. Users can define the SDC checking frequency using a configuration file. Higher frequency decreases the detection latency, and lower frequency minimizes the computational overhead. The evaluation is divided in several subsections that look at different aspects of data corruption and SDC detection.

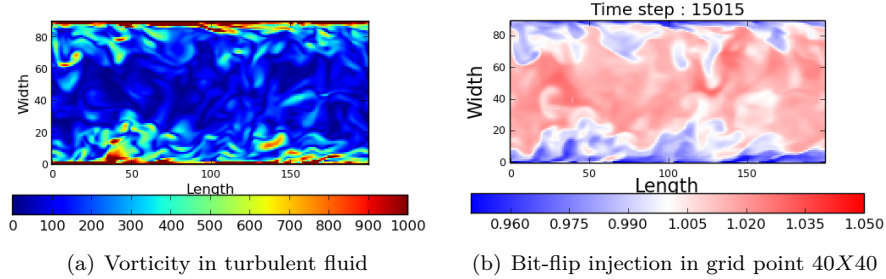


Figure 2: Study of corruption on a turbulence code for compressible gas

#### 4.1 Corruption Propagation

We start by analyzing how corruption propagates in classic HPC applications, such as the CFD code mentioned above. Most CFD applications produce vorticity plots that show the level of turbulence of the fluid. For instance, Figure 2(a) shows the vorticity of the fluid on a 2D cut of the 3D duct, aggregating data from all MPI processes in the simulation (total length of the duct). However, the vorticity is computed from the velocity fields in the three axes (for visualization) and is never stored in a variable. Therefore, an error deviation observed in the vorticity plot is likely to be the consequence of a corruption happening in one of the velocity fields.

Figure 2(b) shows the velocity field following the x axis ( $u_1$ ). As we can see, velocity changes close to the walls of duct as a result of the viscosity of the fluid. In this run, we injected a bit-flip (grid point 40X40) in the 24th bit position, the first bit of the exponent. This corruption is barely visible as a tiny white dot in the middle of a red area. Yet this corruption will generate large perturbations that will propagate across the domain, reaching other MPI ranks and corrupting the large majority of the domain.

To study the propagation of corruption across after an SDC, we performed the following experiment. First, we launched a turbulent flow simulation starting from the initial conditions and let it run for 15,000 time steps, while checkpointing every 500 time steps, with the last checkpoint taken at iteration 15,000. The purpose was to let the gas reach a relatively high level of turbulence. Then, we restarted the execution from the last checkpoint (i.e., time step 15,000), and we recorded the datasets of the execution at each time step for a corruption-free execution. We repeated the same corruption-free execution several times and

confirmed that for each time step, all datasets are identical between executions. Then, we repeated the same experiment (restarting from time step 15,000) but this time injecting one bit-flip at bit position  $p$  for  $p$  in 10, 12, 14, 16, 18, 20, 22, 24. For each experiment we injected the bit-flip in the first twenty time steps (i.e., before time step 15,020) and let it run for 500 iterations.

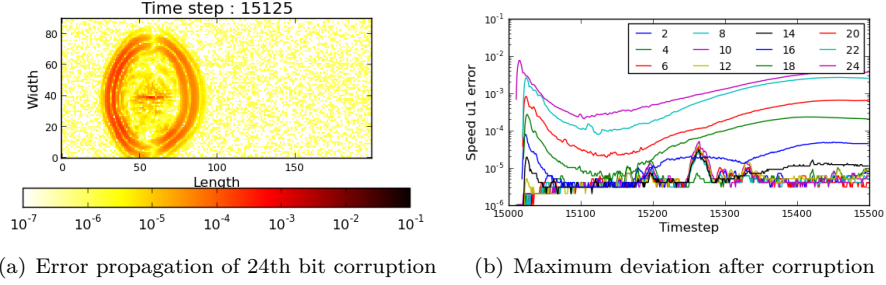


Figure 3: Error propagation in turbulent flow simulation

After all the corruption experiments were done, we computed for each experiment and for each time step the difference between the corrupted dataset and the corruption-free dataset. Figure 3(a) plots this deviation a hundred time steps after corrupting the 24th bit of the grid point 40X40. We use a logarithmic color scale to show the magnitude of the deviation in the different regions of the domain. For this simulation, we set the error tolerance to  $10^{-6}$ , meaning that any error lower than  $10^{-6}$  will be ignored. As we can observe, in only a hundred iterations the corruption has already propagated across the entire domain, and it shows a particularly high deviation in a region with a wave shape that has as origin the corrupted grid point. Although the origin of the corruption was in the grid point 40X40, the fluid has moved in those hundred time steps, and the corruption wave has as epicenter about 20 grid points to the right, which happens to be located in an MPI rank other than the one where the corruption was originally injected.

Looking at the following hundred time steps, we noticed that the high deviation wave bounces in the walls of the duct and continues propagating in other directions. We plotted similar figures for each time step of each corruption experiment, but here we show only one for brevity. To get an idea of how deviations behave for different corruption levels, we plotted the maximum deviation at each time step for all the corruption experiments. The results are shown in Figure 3(b). As we can see, during the first ten time steps or so, there is no corrupted data. When the bit-flip is injected, we observe a sudden jump with a magnitude exponentially proportional to the bit-flip position, which is consistent with the floating-point representation. In addition, we notice that immediately after the corruption jump, the deviation starts to decrease. This decrease is due to a *smoothing* effect that takes place when the noncorrupted data interacts with the corrupted data. However, the same influence can go in the other direction. For instance, when the corruption wave bounces in the wall

of the duct, it interacts with the other part of the wave that is just arriving to the wall, generating a corruption amplification effect, which is what we see happening after time step 15,150. Finally, the deviation stabilizes around time step 15,400 and remains stable until the end of the execution.

## 4.2 Data Distribution of the Detectors

Having a better idea of how corruption propagates, we move to the next part of our evaluation, namely, the study of the PDF of the different detectors (see Section 3). We first analyze the PDF of each detector at a given time step  $k$  for a corruption-free execution. The goal of this study is to visualize the distribution for each detector, compare with previous observations, and predict the possible detection impact of each detector.

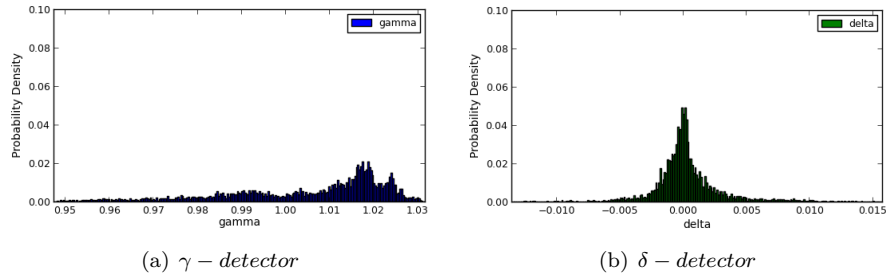


Figure 4: Probability density function of gamma and delta detectors.

The first detector we study is the  $\gamma$  - *detector*. This detector is simply the PDF of the target dataset. Figure 4(a) shows the PDF for the  $u1$  velocity field. As we can see, most of the values are higher than 1.0, which is consistent with Figure 2(b). A smaller percentage of the dataset shows values under 1.0, which corresponds to the areas close to the duct walls where the speed decreases. The shown distribution uses 256 bins in single precision, for a total size of 1 KB of data. From the plot we notice that the data is relatively well distributed across the spectrum of values and that none of the bins holds more than 2% of the data points. Given the statistical dispersion of the data and the wide interdecile range (IDR), the  $\gamma$  - *detector* is not expected to perform well for this dataset.

The second detector we analyze is the  $\delta$  - *detector*. The PDF produced by the  $\delta$  field is shown in Figure 4(b). We can see a clear Gaussian distribution with 0 as the expectation and a low variance. We note that the dispersion of the  $\delta$  distribution is low and its IDR is several times smaller than the IDR of the  $\gamma$  distribution. Hence, it is more sensitive to data corruption. Thus, for this particular dataset, the  $\delta$  - *detector* is expected to perform better than the  $\gamma$  - *detector*, yet some rare anomalies might be detected by the latter and ignored by the former.

The PDF produced by the  $\epsilon$  - *detector* also shows a Gaussian distribution, although with a slightly different shape from that of the  $\delta$  distribution. Again,

the expectation is 0.0, and the variance is low. We note that the  $\epsilon$  IDR is about one order of magnitude smaller than the IDR of the  $\delta$  distribution. This is due to the high time resolution, low space resolution, and the particular characteristics of the fluid (e.g., viscosity). Other executions of a similar CFD simulation could produce much different distributions by simply changing some input parameters.

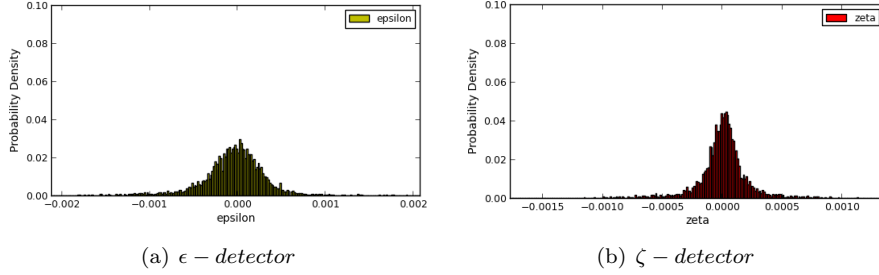


Figure 5: Probability density function of gamma and delta detectors.

The  $\zeta$  - detector also produces a Gaussian distribution. The IDR is within the same order of magnitude but is narrower than previous cases. Thus, for this dataset it is expected to show the highest recall among all four detectors.

### 4.3 Clustering and Outlier Detection

Now that we have a clear idea of the distribution of each detector, let us see how this can be used to detect SDC. We run the simulation together with the four detectors, which are checking for data corruption at every time step. Then, at time step  $t^k$ , we inject multiple bit-flips in different points of the domain and at different bit positions for each injection. Just after the bit-flips have been injected and the detectors have analyzed the datasets, we plot the detectors' state as a couple of two-dimensional clouds of points where it is easy to visualise the outliers. We color each point depending on the level of corruption of each particular point (dark blue means no corruption).

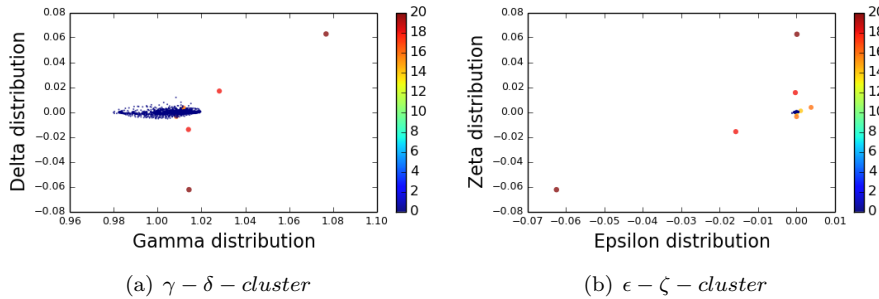


Figure 6: Outlier detection with multiple detectors.

As Figure 6(a) indicates, corruptions in the high bits of the mantissa (i.e., bits 18 and 20) generate clear outliers that are at a large distance from the  $\gamma - \delta - cluster$  of noncorrupted points. We note that some corruptions are detected multiple times, once when analyzing the corrupted point and more when analyzing the points around the corrupted one. For instance, in Figure 6(a) we see two corruptions in the upper-right part of the plot. These correspond to the actual corrupted data points because both  $\delta$  and  $\gamma$  are distant from the cluster. In the bottom part of the plot, we observe two corruptions complementing those two outliers. These are neighbors that have noncorrupted data (normal  $\gamma$ ) but that do see a strange distance (abnormal  $\delta$ ) from their corrupted neighbor. This interesting feature of the detection mechanism can help increase the confidence in a given outlier detection. However, we see that this does not help increase the detection of less critical corruptions (i.e., bits 16 and 14). Such corruptions are located within the  $\gamma - \delta - cluster$  and are not detected as outliers by either of these two detectors.

Fortunately, the  $\epsilon - \zeta - cluster$  does a much better job, positioning both corruptions outside the cluster. In fact, the corruptions on the high bits of the mantissa generate such a large deviation that the cluster looks as a pale blue dot, as shown in Figure 6(b). We also observe the same multiple-detection phenomena as in the previous case. The reason is that the  $\zeta - detector$  is also based on a spatial analysis, making neighbor points capable of detecting the corruption. Thanks to the visualization of the detectors' distribution as clusters, one can now easily understand the fuzzy logic module and the scoring system. The distance from the point to the cluster quantifies the level of deviation for each detector. Combining the weighted score of the four detectors gives the final outlier score that is reported to the user.

#### 4.4 Detection Recall

The next step in our evaluation is to inject bit-flips during runtime and see how our detectors react. Thus, we run an experiment in which corruptions are injected at random points in time, at random positions in the domain and at random bit positions. All injections are logged in order to compare the injections and the detections with a post-mortem script. For the first experiments we use the already familiar turbulent flow code. We set the allowed error to  $10^{-5}$  so corruptions happening in the last 4 bits of the mantissa can be neglected (see Figure 3(b)). For fairness, we do not take into account corruptions on the 8 most significant bits because, although our detectors can detect them, those corruptions make the application unstable and crashes. As shown in Figure ?? our detectors can notice perturbations as small as those affecting bit  $15^{th}$  of the floating point representation. Thus, this technique detects about 50% of the vulnerable bits, giving an overall coverage of about 68.75%.

We repeat the same experiments with a cosmology application: HACC. HACC is an n-body simulation that is helping scientists understand the role of black matter in the universe. The application produce results that are then studied using statistical tools, which makes the lose of precision of a few particles

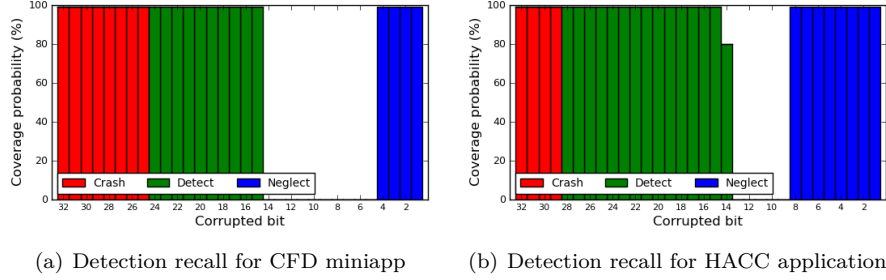


Figure 7: Detection recall study on HPC applications.

less dramatic. Due to this reason, we set the negligible corruption threshold to bit position  $8^{th}$ . For the same reason, the application is in general more stable and even corruptions on the exponent bits will not produce a crash. On HACC, our detector is able to notice perturbations in 14 out of the 20 remaining vulnerable bits, which is a recall of 70%. This translates into an overall coverage of 80%.

## 5 Conclusion

In this work we have studied some of the properties of HPC applications and their datasets. Based on these properties, we proposed several SDC detectors that scrub the datasets of an application and generate the next expected distribution in order to detect anomalies. To integrate all the detectors together, we proposed a fuzzy logic module that gives detailed information to the user about the detected anomalies. We implemented these lightweight algorithms and minimized their memory footprint using an indexing strategy.

We evaluated our proposed scheme with a turbulent flow simulation based on Navier-Stokes equations. We studied the propagation of corruption across the entire domain and for different levels of corruption. We analyzed the distributions generated by our four detectors and demonstrated their detection accuracy using a cluster visualization technique. The results show that our technique can detect the majority of corruptions while incurring only negligible overhead on the scientific application. We demonstrated that with the help of these detectors, applications can have up to 80% of coverage against data corruption.

As future work, we would like to improve the prediction of the PDF evolution using a Kalman filter. We also plan to implement subregion decomposition inside each MPI rank, in order to increase further the detection recall of our SDC detector.

## Acknowledgments

This material is based upon work supported by the U.S. Department of Energy Office of Science, under contract number DE-AC02-06CH11357.

## References

- [1] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *SC*, page 32. ACM, 2011.
- [2] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *International Journal of High Performance Computing Applications*, page 1094342014532297, 2014.
- [3] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25:10–16, November 2005.
- [4] A. Cataldo. Mosys, iroc target ic error protection, 2002.
- [5] Z. Chen. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 167–176. ACM, 2013.
- [6] T. J. Dell. A white paper on the benefits of Chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division*, pages 1–23, 1997.
- [7] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.
- [8] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.
- [9] D. Li, J. S. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 57. IEEE Computer Society Press, 2012.
- [10] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture.*, pages 243–247. IEEE, 2005.
- [11] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, 1996.
- [12] T. Semiconductor. Soft errors in electronic memory - a white paper, 2004.



## **Mathematics and Computer Science Division**

Argonne National Laboratory  
9700 South Cass Avenue, Bldg. 240  
Argonne, IL 60439-4847

[www.anl.gov](http://www.anl.gov)



Argonne National Laboratory is a U.S. Department of Energy  
laboratory managed by UChicago Argonne, LLC